

# Algoritmos y Estructuras de Datos 2

Final 2022-08-09

## Ejercicio 1

- a. **Falso.** Una operación es un observador básico si permite observar alguna característica distintiva de la instancia del TAD, y junto al resto de los observadores básicos, poder determinar si 2 instancias son observacionalmente iguales o no. Cualquier operación puede ser axiomatizada sobre los generadores, eso no determina si la operación es un observador.
- b. **Verdadero.** Significa que esa operación está observando alguna característica necesaria para determinar la igualdad observacional. O dependiendo del contexto de uso, también se puede eliminar esa operación del TAD para solucionar el problema.
- c. **Falso.** Asumiendo que el TAD está bien definido, si 2 instancias del TAD son observacionalmente iguales esto no puede suceder.
- d. **Verdadero.** El enunciado pide que B sea un comportamiento automático. Como el nombre lo indica, los efectos de B deben suceder automáticamente cuando sucede A. Esto se debe modelar durante la axiomatización de A, ya que si tenemos una operación explícita para B, estamos delegando al usuario la responsabilidad de aplicar B siempre que suceda A (y el usuario puede no hacerlo por olvido o intencionalmente).

## Ejercicio 2

- a. **Falso.** El invariante debe valer en la Pre y Post únicamente en las funciones de la interfaz, es decir, las funciones públicas del módulo. En las funciones auxiliares podemos romper el invariante, e incluso es muy común tener funciones auxiliares que reestablecen el invariante de representación (por ejemplo una función para balancear un AVL).
- b. **Falso.** Los algoritmos de las operaciones se definen a partir del invariante de representación, ya que éste tiene que valer en la Pre y Post y nos garantiza propiedades de la estructura y también nos obliga a mantenerlas al finalizar la operación. Además, el invariante probablemente haya sido diseñado antes que los algoritmos.
- c. **Verdadero.** El invariante determina únicamente la **cota inferior** de la complejidad de las operaciones. Por ejemplo, el invariante del AVL nos garantiza ciertas propiedades sobre la estructura: cada nodo es un ABB y el factor de desbalanceo  $F$  es tal que  $|F| \leq 1$ . Gracias a esto, podemos implementar algoritmos que se apoyan en estas propiedades para ser eficientes, en el caso del AVL lograr las operaciones en  $O(\log(n))$ . Aún así, nuestros algoritmos pueden ignorar estas propiedades y ser muy ineficientes, por eso el invariante no determina una cota superior.

## Ejercicio 3

- a. **Verdadero.** Estas técnicas asignan a las operaciones un costo mayor del que realmente tienen en concepto de “crédito a favor” o “potencial acumulado” a ser usado en futuras operaciones. De esta forma, cuando analizamos la ejecución de múltiples operaciones a lo largo del tiempo, podemos realizar un promedio aritmético de la complejidad para determinar la complejidad promedio de una operación en particular.
- b. **Verdadero.** Si suponemos que todas las operaciones individuales tienen una complejidad del peor caso, al calcular el promedio éste va a ser el peor caso, pero nunca peor que eso. Las operaciones que sean mejores que el peor caso van a mejorar el promedio.
- c. **Falso.** Las skip lists perfectas garantizan  $O(\log(n))$  para la búsqueda pero la inserción en el peor caso resulta  $O(n)$  si se inserta un elemento al inicio de la skip list (hay que reacomodar todos los niveles del resto de los elementos). Las skip lists randomizadas tiran una moneda repetidas veces hasta que salga cruz, y la cantidad de caras obtenidas indica hasta qué nivel debemos conectar el elemento insertado. En el caso promedio esto logra una complejidad de  $O(\log(n))$  para ambas operaciones, pero en el peor caso serían  $O(n)$ .
- d. **Falso.** En el peor caso es  $O(n)$  si insertamos en la primer página que está llena y ahora necesitamos rearmar todas las páginas siguientes.

## Ejercicio 4

- a. **Falso.** Sea  $f = n^2$ ,  $g = n^3$ . Se verifica que  $f = n^2 = O(n^3) = O(g)$  y también que  $g = n^3 = \Omega(n^2) = \Omega(f)$ . Pero no vale que  $f = n^2 \neq \Theta(n^3) = \Theta(g)$  ya que no se puede acotar  $f = n^2$  por arriba y por abajo  $\pm$  constantes con  $g = n^3$ .
- b. **Verdadero.** Sea  $f = n$  la complejidad del mejor caso del InsertionSort (cuando la entrada ya está ordenada y por lo tanto el ciclo interno no tiene que hacer nada y solo recorremos 1 vez el arreglo con el ciclo externo). Suponiendo ahora una entrada arbitraria, se verifica que el algoritmo tenga complejidad  $\Omega(n)$ , pues el mejor caso define la cota inferior de complejidad. Para una entrada arbitraria, si ésta resulta ser un caso promedio o el peor caso, el algoritmo puede resultar más ineficiente pero la cota inferior sigue siendo válida.
- c. **Falso.** Siguiendo con el ejemplo del InsertionSort, sea  $f = n^2$  la complejidad del peor caso (cuando la entrada está ordenada al revés). Supongamos que la entrada es ahora el mejor caso, donde ya vimos que la complejidad es  $O(n)$ . Por definición, un algoritmo tiene complejidad  $\Theta(f)$  si se puede acotar por arriba y por abajo con  $f$ , o dicho de otra forma, si el algoritmo es  $\Omega(f) = O(f)$ . Pero como hemos visto en el ejemplo, InsertionSort es  $\Omega(n)$  en el mejor caso y  $O(n^2)$  en el peor, por lo tanto el enunciado es falso.
- d. **Falso.** La complejidad del mejor caso de un algoritmo para un cierto problema es mayor o igual que cualquier límite inferior para el problema. Por ejemplo, cualquier algoritmo de ordenamiento basado en comparaciones tiene demostrado que la cota inferior es  $\Omega(n \log(n))$ . Esto significa que la complejidad óptima que podemos lograr con este tipo de algoritmos es  $\Omega(n \log(n))$ , no obstante puede haber (y hay) algoritmos del mismo tipo pero más ineficientes.
- e. **Falso.** Sean  $S = [7, 3, 6, 1, 2, 4, 5]$ ,  $i = 4$ ,  $j = 5$ . Veamos que  $i < j$  y que  $S[i] = 2 < 4 = S[j]$ . Si intercambiamos  $S[i]$  por  $S[j]$  se rompe el heap.
- f. **Falso.** Sean  $S = [7, 3, 6, 1, 2, 4, 5]$ ,  $i = 2$ ,  $j = 4$ . Veamos que  $i < j$  y que  $S[i] = 6 > 2 = S[j]$ . Si intercambiamos  $S[i]$  por  $S[j]$  se rompe el heap.